A Directed Graph Framework Implementation for the Euclideus Language

Justin Gagnon

July 11, 2017

The present manual introduces a framework to the Euclideus language for building arbitrary directed graphs, which may be used to implement data structures such as lists, trees, networks, and more. This framework is centered on a new type of native object: the `node` type. The following manual presents a full specification for the syntax and ideas pertaining to `node` objects, with a few examples.

CONTENTS

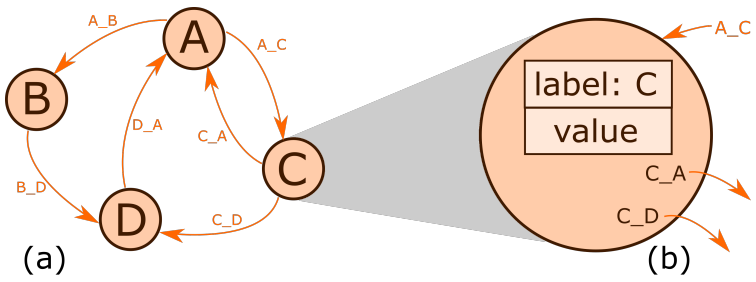

Figure 1.1: In a typical directed graph, shown in panel (a), edges (represented by arrows) connect nodes (represented by the circles) to other nodes. As displayed in panel (b), nodes contain (i) two objects – the node's label (its variable name) as well as a value which is an object of any type, and (ii) a set of outgoing edges. All edges are properties *only of the nodes they originate from*, thus node `C` contains edges `C_A` and `C_D`, but not edge `A_C`, which belongs to node `A`. All edges, while properties of their source nodes, are not actual objects themselves.

1 Directed Graphs and The Node Object

Directed graphs are used in various scenarios, including flow charts, state diagrams, commutative diagrams and data structures. A directed graph is mathematically defined as a set of *nodes* which are connected to each other through a set of *edges*. In Fig. 1.1-a, the nodes are represented as circles and the edges are arrows between the nodes. The fact that each node-node connection has a direction is what makes such a graph a *directed* graph.

In the present framework, only the `nodes` themselves are objects while edges are merely properties of a node – specifically, an edge `E` going from some `node N` to another `node` is a property of the `node N`. This aspect is illustrated in Fig. 1.1-b. As an object, a `node` contains only two members, which are both private.

First, each `node` contains a label which is a `primitive` object representing an identifier unique to the `node`. The `node`'s label coincides with its "variable name", i.e. the name of the node object as stored in the local environment. This label may alternatively be viewed as its address in memory. `nodes` also encapsulate data in a member called `value`, by default public, which may be an object of any type, and is mutable.

Unlike all other object types in the Euclideus language, `node` objects contain an additional set of properties which are its outgoing edges. These edges are represented as the arrow symbols in Fig. 1.1. A `node`'s edges are not objects themselves, they contain only a single piece of information which is the `node` they point to. Each edge has a name, and edges may be dynamically added to or deleted from a node. As a last note, edges can *only* point to other `node` objects (or to nothing at all), such that `nodes` can only connect to other `nodes`, not to objects of other types.

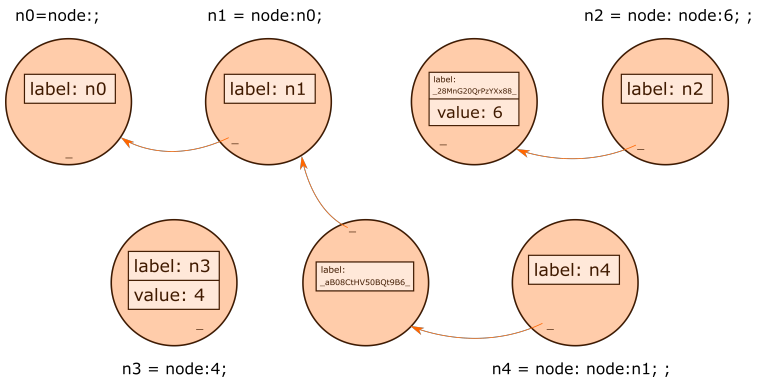

Figure 2.1: The above figure represents the result of repeated calls to the `node` object constructor, corresponding to example (2.1)-(2.5)

Finally, as a matter of nomenclature let us identify two special cases for `node` objects: (i) if a `node` does not contain a `value` member, it is called an *empty* `node`, and (ii) if a `node` contains no outgoing edges pointing to other `nodes`, it is called a *leaf* `node`; in other words, a leaf `node` may be viewed as a "dead-end" in a graph. Note that a leaf `node` may still contain a `value`.

2 Node Syntax

Objects of `node` type are native to the system, they are not associated to a class and rely on certain syntactical elements that are specific to their particular type, namely for operations involving edges (which are entities that never exist on their own outside of a `node`).

2.1 Node Construction

A `node` `n` can be initialized by calling the `node` object constructor, using the familiar constructor syntax. The `node` constructor may be called with either one argument or no arguments at all. If no arguments are passed to the `node` constructor, as in (2.1), a simple node is created with no `value` and with a single edge called _ (underscore)[1] which does not point to any `node` object, as illustrated by `node` `n0` in Fig. 2.1. Such a `node` containing no value and no outgoing edges is referred to as a *leaf* node. Whenever a node is assigned to a variable name, its `label` member automatically becomes that variable name. A `node` object's `label` member and its variable name are always identical.

Below is a set of examples showcasing the behaviour of the `node` constructor for various parameter types.

[1] By default, the `node` constructor always creates a node with an edge called _ (underscore).

```
n0 = node:;                                  (2.1)
n1 = node:n0;                                (2.2)
n2 = node:  node:6; ;                        (2.3)
n3 = node:4;                                 (2.4)
n4 = node:  node:n1; ;                       (2.5)
```

If an argument is passed to the `node` constructor, the constructor's behaviour depends on the parameter's type, specifically whether or not it is itself a `node` object. If the parameter is not a node, as in (2.4), then it is encapsulated inside a new `node` object – as the `node`'s `value` – with the resulting `node` being assigned to the LHS of the assignment statement (the `node`'s label thus becomes the assigned variable name). Again, the new `node` contains a single edge called `_` pointing to nothing. This case is illustrated by the `node` `n3` in Fig. 2.1

Now, if the argument passed to the constructor is itself a `node` object, as in (2.2) and (2.3) and (2.5), then the constructor does not encapsulate the parameter as the `value` of the newly created `node`. Instead, it creates a new *empty* `node`, and connects its default edge (i.e. the underscore `_` edge) to the `node` passed as a parameter to the constructor. In this regime it is useful to distinguish between two cases: whether or not the input `node` (the one passed as a parameter) already exists in the environment. If the input `node` does already exist, in other words if it was previously created and assigned to a variable name in the environment as in (2.2), then the constructor creates an empty node and points its default `_` edge to the input `node` parameter. This case corresponds to `node` `n1` in Fig. 2.1.

On the other hand, if the `node` object passed as a parameter to the constructor does not already exist, e.g. (2.3) where the constructor is invoked with a `node` parameter which itself was just constructed, then *two* new `node` objects are created and assigned in the environment. In the first step, the interpreter builds a `node` for the input parameter. In (2.3), the parameter is defined using the constructor with a `primitive` value of `6`. Thus, the node created from `node:6;` encapsulates `6` as its `value` and the interpreter automatically generates a `label` for it, which is an alphanumeric case-sensitive character string bounded by underscore characters, chosen so as to preclude collisions with any of the previously assigned variables in the environment. Such a `node` is represented by the one called `_28MnG20QrPzYXx88_` in Fig. 2.1 (the second from top-right). Having created and assigned the input `node`, the assignment statement (2.3) may now be carried out by generating an empty node, assigning it to `n2`, and pointing its default edge `_` to the newly created input `node`, as illustrated by `node` `n2` in Fig. 2.1.

As a last example, let us consider case (2.5). Here, the `node` constructor is called with a parameter which is a hitherto nonexistent `node`, being an invocation of the constructor passed with an existing `node` (`n1`). In this case, the instruction `node:n1` creates an empty `node` (called `_aB08CtHV50BQt9B6_` in Fig. 2.1), connecting its `_` edge to the pre-existing `n1`. This newly created `node` is then passed as an argument to the outer constructor thus creating another

empty node n4 and connecting it to the node _aB08CtHV50BQt9B6_.

2.2 Edge Syntax

In order to access a node object's properties – namely its functions and its edges (nodes do not contain any public data members) – Euclideus employs the usual indirection mechanism based on the . operator. Since the node object's edges are not objects themselves, the aim of this section is to explain the concepts pertaining to the edge syntax.

2.2.1 Traversing a Directed Graph by Indirecting to Edges

Graph traversal in Euclideus is achieved by indirecting to one of a node's edges, which yields the node targeted by the edge in question. To understand this more clearly, consider the example (2.1-2.5) presented above, as illustrated in Fig. 2.1, specifically the linear graph formed by the four connected nodes n4 → _aB08CtHV50BQt9B6_ → n1 → n0. Starting from n4, the expression n4._ yields the node object connected to n4 via the edge _, which is _aB08CtHV50BQt9B6_. Likewise, n4._._ therefore yields a node two hops away from n4, connected by the edge _ of n4 and edge _ of _aB08CtHV50BQt9B6_, resulting in node n1. Similarly, n4._._._ will return node n0.

If an edge-indirection expression cannot successfully traverse a graph – either because an edge does not exist or because it is not connected to a node – then a new empty leaf node is returned (recall that a leaf node is one which does not connect *to* another node. Thus for instance, the following expressions all produce leaf nodes:

```
n0._                    (2.6)
n4._._._._              (2.7)
n4.edge1                (2.8)
n4.edge1.edge2          (2.9)
```

The first two indirections (2.6) and (2.7) yield leaf nodes because the edge _ of n0 does not connect to any node, while (2.8) and (2.9) contain indirections to nonexistent edges edge1 and edge2.

Edge identifiers are interpreted as string literals. As such, they may contain *any* of the characters in the lower ASCII range (except for the the null character); this includes all lower- and upper-case latin letters, numerals, space and special characters. Since some of these characters – e.g. +, -, -, etc. – are understood as operators by the interpreter, an edge identifier containing these characters should be enclosed inside quotation marks ("). Moreover, it is important to note that edge identifers are always pre-evaluated as strings. The following example will clarify and illustrate these details.

```
  n=node:0;                                  (2.10)
n.s=node:1;                                  (2.11)
  s="$pec!@/ chars^"                         (2.12)
n.s=node:2;                                  (2.13)
```

In the above example, (2.10) first initializes a `node` `n` with a `value` of `0`. Next, on (2.11) an edge called `s` is created in `n` and pointed to a new `node` containing a `value` of `1`. Now, a `string` object `s` containing special characters and a space is defined on line (2.12). Consequently, having assigned a `string` to the identifier `s`, the left-hand side of (2.13) *no longer* evaluates to the edge `s`, but rather to `n."$pec!@/ chars^"`, since that `string` was ascribed to the identifier `s` in (2.12). Thus, (2.13) creates a second edge inside `n`, called `"$pec!@/ chars^"`, and points it to a new `node` with `value` 2. In order to access the edge `s` inside `node` `n`, it is necessary to place the identifier inside quotation brackets: `n."s"` yields the `node` with `value` `1` while `n.s` yields the `node` with `value` 2.

This particular syntax may be exploited to automate the generation of multiple outgoing edges from a `node` using just a few lines of code. Using the `node` `n` in (2.10-2.13), consider the following loop:

```
for:j=0,j<1000,j=j+1,                        (2.14)
   n.(j.print())=j                           (2.15)
;                                            (2.16)
```

Here, `1000` new edges are generated inside `n`, labeled with the index `j`, each of them pointed to a new `node` containing `value` `j`. Note on (2.15) that the left-hand side of the assignment creates a `string` object from `j` (a `primitive` type) via the `print()` function in order to produce a unique edge, otherwise `n.j=j` would simply reassign a new `node` to the same edge `"j"` at each iteration of the loop.

2.2.2 Creating and Connecting Nodes Using the Edge Syntax

In addition to enabling graph traversal, edges may also be used for creating and connecting `nodes` dynamically, performed via an assignment expression. To show how this is done, let's begin with initial `nodes` defined by the assignments `nodeA = node:"A";` and `nodeB = node:"B";`. This creates two `nodes` `nodeA` and `nodeB` containing string values of `"A"` and `"B"`, respectively, each having a default edge _ pointing to nothing. Now consider the following assignment expressions

```
nodeA.newEdgeToB = nodeB                          (2.17)
nodeA._ = node:"Hello!";                          (2.18)
nodeA.newEdge1 = 1                                (2.19)
nodeA.newEdge1.newEdge2 = 2                       (2.20)
nodeB._ = nodeA.newEdge1.newEdge2                 (2.21)
```

Each one of the assignments (2.17-2.21) is responsible for creating a connection between nodes. The LHS of each assignment is an indirection to an edge in an existing node while the RHS is an object (of node type or otherwise).

In the first assignment (2.17), a new edge called newEdgeToB is created in nodeA and linked to the existing nodeB. In the second assignment (2.18), nodeA's default edge _ is connected to a new node object created with the value "Hello!". Thus, this assignment statement performs two actions: it (i) creates a new node and (ii) connects an edge to it. These first two examples are assignments to node-type objects explicitly.

By definition, since edges can only connect to node objects, Euclideus offers a syntactical shortcut for node connections: If the RHS of the assignment is not a node object, then the object on the RHS is automatically encapsulated into a newly created node. Thus, in (2.19) a new edge newEdge1 is created in nodeA and attached to a new node constructed with the value 1. Similarly, in (2.20) an edge newEdge2 is created inside the latter node (the one containing value 1) and is attached to a new node containing value 2.

In (2.21) nodeB's default edge _ is connected to the existing node nodeA._. Whenever the RHS of an edge connection assignment evaluates to an existing node, no new node is created. On the other hand, if the RHS of an edge connection assignment is a non-node or a non-existing node – such that the edge on the LHS targets a newly generated node – then the new node's encapsulation attribute, i.e. whether the node is private or public, becomes that of the edge's parent node. Thus, if nodeA were private, then nodeA.newEdge1 and nodeA._ would also both be private since these are new nodes generated as a result of edge assignments (2.18) and (2.19); the same argument applies to nodeA.newEdge1.newEdge2. In contrast, nodeB would retain its original encapsulation attribute since it was already created priot to the edge connection (2.17).

Lastly, note that the LHS of an edge connection assignment must always indirect an edge of an existing node, in other words the LHS of such an assignment is never used to create new nodes. For example, the instruction nodeA.newEdge3._ = 3 is impossible to execute because there exists no node as of yet connected to newEdge3 in nodeA, which is to say that nodeA.newEdge3 is a non-existing node.

The directed graph resulting from (2.17-2.21) is displayed in Fig. 2.2.

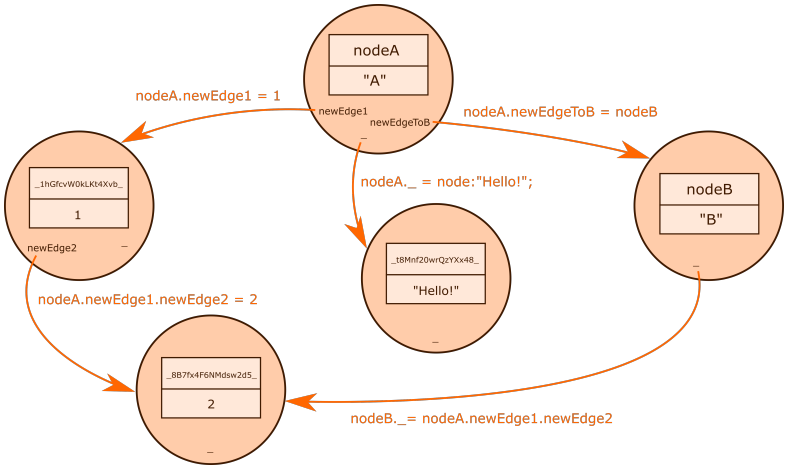

Figure 2.2: The above figure represents the directed graph resulting from executing instructions (2.17-2.21), where two nodes (`A` and `B`) were initialized in the standard manner via the `node` constructor, and subsequent `nodes` were created automatically by assigning edges to new `nodes`.

3 Node Functions

Each `node` object is equipped with a set of built-in member functions. These are detailed below.

3.1 Getting a Node's Label

Each `node` object has a getter function for retrieving a representation of its `private label` member. Given a `node n`, a call to `n.label()` (a function taking n arguments) will return a `string` object representing the `node`'s label (its name), which is obviously `n` in this case. Note that the `label` member is of `primitive` type while the `label()` function returns a `string`.

The `label()` function is most useful when invoked on `nodes` that were automatically created by the system – such as those created in (2.17-2.21) whose labels contain sixteen alphanumeric characters. In that example, the `string` comparator

```
nodeA.newEdge1.newEdge2.label().compare(nodeA.newEdgeToB._.label())   (3.1)
```

evaluates to `0` since the paths `nodeA.newEdge1.newEdge2` and `nodeA.newEdgeToB._` lead to the same `node`, and hence both calls to `label()` return the exact same `string`.

3.2 GETTING A NODE'S VALUE

A node's value, by default a public member, may be accessed simply by indirecting to it. Given a node n, n.value will produce the object stored as n's value member, which can be reassigned a new object with the usual syntax, e.g. n.value=5.

On the other hand, if a node is locked (see below), its value is private, which makes it unmodifiable and directly inaccessible. Thus a getter function value() function, taking no arguments, is implemented to access a locked node's private value. For a node n, a call to n.value() will return *a copy* of its value member if it exists, otherwise nothing is returned. Thus, according to example (2.17-2.21), the boolean expression

For the sake of efficiency, if a node n is unlocked then it is recommended to get its value directly, as n.value, instead of calling the n.value() getter function, which always returns a copy of the value.

```
nodeA.newEdge1.value()+1 == nodeA.newEdgeToB._.value()          (3.2)
```

returns 1 (true) – the path described by nodeA.newEdge1 leads to a node with value 1 while nodeA.newEdgeToB._ leads to a node with value 2. If a node n does not contain a value (e.g. n=node:;) then the result of n.value() is a string representing the node's label (which is exactly the same as a call to n.label()).

3.3 TESTING IF A NODE IS A LEAF OR IS EMPTY

When traversing a graph it is often useful to determine if an endpoint has been reached, in other words a node which does not connect *to* other nodes. The isleaf() function is provided to this end; this function takes no arguments and returns 1 if the node is a leaf node and 0 if it contains outgoing edges to other nodes. As per example (2.17-2.21), nodeA._.isleaf() returns 1 since the associated node (containing the string "Hello!") contains no outgoing edges.

Another type of special case is defined if a node contains no value member. Such a node is said to be empty. In this case, an attempt to call its value() function will result in a string representing its label. Therefore, if a graph can contain nodes without a value then it may be appropriate to check the existence of a value, using the isempty() function, prior to calling the value() function. Given a node n, n.isempty() returns 1 if n contains no value or 0 otherwise.

3.4 DISCONNECTING NODES

The edges between nodes can be removed dynamically via the disconnect() function.For example, in the example (2.17-2.21), a call such as nodeA.disconnect(newEdgeToB) will

remove the edge from nodeA going to nodeB. As edges are always properties of the *source* node, an edge can only be removed by calling disconnect() from its source node.

3.5 Listing a Node's Incoming and Outgoing Edges

While a directed graph is always traversed in the direction of its edges, and only outgoing edges are expressed node properties, nevertheless all nodes are aware of their set of incoming edges (edges originating from other nodes). Both incoming and outgoing edges for a given node can be listed by calling the node's inedges() and outedges() functions, respectively. Both of these functions take no arguments, and return an array of strings representing the edges in question. For the inedges() function, called from some node n, the returned array elements are string representations of the actual syntax used to traverse the graph from the source node to n. On the other hand, the array elements produced by outedges() adopt a particular format.

Keeping with example (2.17-2.21), as illustrated in Fig. 2.2, a call to nodeA.outedges() will yield an array composed of the string elements (in order):

```
"newEdge1 -> _1hGfcvW0kLKt4Xvb_",   (3.3)
"newEdgeToB -> nodeB",              (3.4)
"_ -> _t8Mnf20wrQzYXx48_"           (3.5)
```

The outedges() function therefore just returns a representation of the node's outgoing edges together with their target nodes' labels.

Calling inedges() from nodeA will return an empty array, since there are no nodes connecting to nodeA. However a call to inedges() two hops away from nodeA properly exemplifies this function. The indirected call nodeA.newEdge1.newEdge2.inedges() returns an array of two string elements (in order):

```
"_1hGfcvW0kLKt4Xvb_.newEdge2",   (3.6)
"nodeB._"                        (3.7)
```

3.6 Locking and Unlocking a Node

A node's value is by default publicly accessible and modifiable, however in some cases it may be necessary to render it private, so that it may only be accessible through the getter function value() (described above) and unmodifiable. For this purpose, a lock() function is ascribed to each node object. This function takes a single, or no arguments, and locks the node's value member (if it exists) by rendering it private. If the node is empty (containing no value), then locking it will make it impossible to store a value into it.

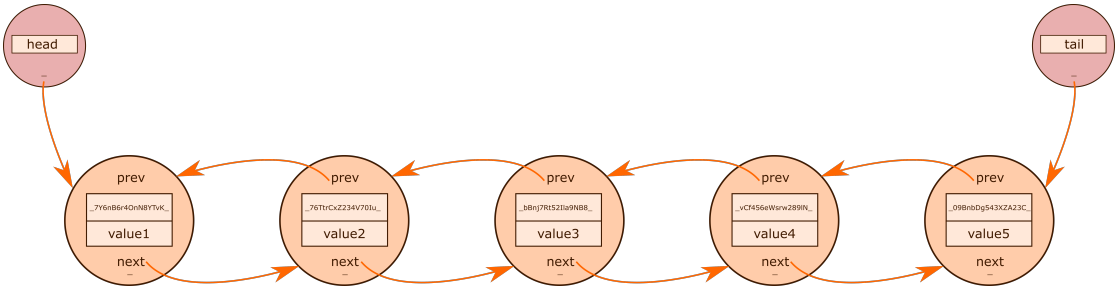

Figure 4.1: A `list` object, instantiated from the `list` class defined herein, is a linear sequence of interconnected `nodes`, bounded by a `head` and a `tail` `node` at its endpoints.

Any argument passed to `lock()` is interpreted as a password `string` which may be later used to unlock the `node`'s value. The password is then hashed and stored as a `private` `primitive`-type `node` member called `password`. The `lock()` function will only have an effect on an unlocked `node`, it returns a value of `1` (true) or `0` (false) depending on whether the `node` was successfully locked or not, respectively.

A second member function `unlock()` is provided to unlock the `node`. Its signature is the same as that of `lock()`: its argument is a password `string`. For the `node` to be successfully unlocked, the password given to `unlock()` must equate to the one given to `lock()`, which will then erase the `private` `password` member and remove the `value`'s `private` attribute (if it exists), rendering it directly accessible and modifiable. The `unlock()` function only has an effect on a locked `node`.

Being objects themselves, `nodes` may also be `private`, say if they are part of a class or another instance object (see the `list` class described below). In this case, they cannot be locked or unlocked from outside of their parent object or class, since `private` members cannot be modified from the outside.

4 A Doubly-Linked List Class Implemented Using Nodes

As a last example meant to synthesize the concepts presented here so far, this section presents an implementation of a doubly-linked list class which leverages the `node` infrastructure. Doubly-linked lists are generally useful in various applications seeing as they can be used to implement stacks and queues, among other data structures. A doubly-linked list is a linear sequence of elements, wherein adjacent elements are connected to each other bidirectionally, as in Fig. 4.1.

As per the present `list` class implementation, a doubly-linked `list` object instantiated using the standard constructor syntax

```
dl_list = list:;                                    (4.1)
```

which initializes a `list` object containing two empty leaf `nodes`, labeled `head` and `tail`. The instantiated object also contains a number of function members used for inserting and removing `nodes` from the endpoints of the `list`. The `head` and `tail` `nodes`, which are always empty and connected to the first and last `node` of the `list` (respectively, are both `private`. Therefore, getter functions `head()` and `tail()` are provided in order to make use of them. A `private size` member is also included along with its getter function `size()`; it keeps track of the number of `nodes` in the `list` (not counting the empty `head` and `tail` `nodes`) such that `dl_list.size()` returns the number of non-empty `list` `nodes`. As depicted in Fig. 4.1, note that a `list` with more than one element contains no leaf `nodes`.

4.1 Inserting Elements Into a List

Elements the `list` class instantiation instructions permit items to be inserted at either side of the `list` using the `insertHead()` and `insertTail()` functions, each taking a single argument: the `value` to be inserted. Upon insertion, a new `node` is created to encapsulate the inserted `value`, and stored at the `head` or `tail` of the `list` (depending on which insertion function was invoked; these functions have no further effect once the `list` is empty.
As an example, the following instructions insert three elements into the initially empty `dl_list`:

```
dl_list.insertHead(array:10;),                      (4.2)
dl_list.insertHead("second value"),                 (4.3)
dl_list.insertTail(22/7)                            (4.4)
```

The first insertion (4.2) creates a `node` encapsulating an `array` of 10 elements and points the default `_` edges of the `head` and `tail` `nodes` to the newly inserted `node`. The second command (4.3) inserts another value (the `string` `"second value"`) at the `head` side of the list, again by encapsulating the `value` in a new `node` and placing it in between `head` and the `node` previously inserted in (4.2). As a result, `dl_list.head()._` now leads to the most recently inserted `node` (containing `"second value"`) while `dl_list.tail()._` still leads to the `node` inserted in (4.2). Additionally, `next` and `prev` edges are created between the two inserted `nodes` ensuring that they are connected to each other. Thus, that `dl_list.head()._.next` leads to the last `node` while `dl_list.tail()._.prev` leads to the first one in the `list`.

Lastly, (4.4) inserts a `value` at the `tail` end of the `list`. Consequently, the new `node` (encapsulating `3.142`) is inserted between `tail` and the last `node` of the `list` (this being the `node` inserted in (4.2)), `tail` is now made to point to the newly inserted `node`, and `next` and `prev` edges are created to link the last and second-last `nodes` in the `list`. As a result, `dl_list.head()._.next.next` leads to the last `node`, `dl_list.tail._.prev.prev` leads to the first one, and `dl_list.head._.next` and `dl_list.tail._.prev` lead to the same

node, i.e. the middle one.

Each of the three nodes created through (4.2-4.4) are objects created inside the environment of dl_list – they are in effect members of the this object. Additionally, they are all private members since they were each created as new nodes targeted by private nodes (head and tail).

The list class implementation overrides the default print() function in order to provide a cleaner representation of list objects. As a result, dl_list therefore produces the following output: head -> "second value" (array 10) 3.143 <- tail.

4.2 Removing Elements from a List

A list can be "trimmed" by removing the nodes at its endpoints using the deleteHead() and deleteTail() functions, these take no arguments. For instance, successive calls to dl_list.deleteHead() and dl_list.deleteTail() will remove the nodes pointed to by head and tail, respectively, and leave dl_list with only a single element (the middle one containing an array). A further call to dl_list.deleteHead() or to dl_list.deleteTail() will render an empty list, and subsequent calls to these functions (on an empty list) will have no effect.

```
list=class:

  private:

    head=node:;
    tail=node:;
    size=0;

  size()=size,
  head()=head,
  tail()=tail,

  insertHead(value)=:
    if: size==0,
      head._=node:value;
      tail=head,
    else,
      head._.prev=node:value;
      head._.prev.next=head._,
      head._=head._.prev;
    size=size+1,
    return=this
  ;

  insertTail(value)=:
    if: size==0,
      head._=node:value;
      tail=head,
    else,
      tail._.next=node:value;
      tail._.next.prev=tail._,
      tail._=tail._.next;
    size=size+1,
    return=this
  ;

  deleteHead()=:
    if: size>1,
      tmp=head,
      head._=tmp._.next,
      delete(tmp._),
      size=size-1,
    elseif, size==1,
      delete(head._),
      size=size-1
    ;
    return=this
  ;

  deleteTail()=:
    if: size>1,
      tmp=tail,
      tail._=tmp._.prev,
      delete(tmp._),
      size=size-1,
    elseif, size==1,
      delete(head._),
      size=size-1
    ;
    return=this
  ;

  print()=:
    return=string:head.label()," ->  ";
    sentinel=head,
    while: ~sentinel._.isempty(),
      type=sentinel._.value.type(),
      if: type.compare("primitive")==0,
          return.appendw(sentinel._.value,"  "),
```

```
70            elseif, type.compare("string")==0,
71              return.appendw(sentinel._.value.print()," "),
72            elseif, type.compare("array")==0,
73              return.appendw("array:",sentinel._.value.size(),"; "),
74            elseif, type.compare("matrix")==0,
75            dims=sentinel._.value.dims(),
76            return.appendw("matrix:",dims.0),
77            for:j=1,j<dims.size(),j=j+1,
78              return.appendw(",",dims.j)
79            ;
80            return.appendw("; "),
81            else, return.appendw(type,":; ")
82          ;
83          sentinel._=sentinel._.next
84        ;
85        return=return.appendw("<- ",tail.label())
86      ;
87    ;
```

5 ACKNOWLEDGMENTS

The author gratefully acknowledges Peter Renkel for valuable discussions and contributing fundamental ideas.